An Educational Rigid-Body Dynamics Physics Engine
TJHSST Senior Research Project Proposal
Computer Systems Lab 2009-2010

Neal Milstein

January 26, 2010

Abstract

The goal of this project is to create a rigid-body dynamics physics engine in order to allow physics students to visualize and solve physics models. Rigid-body dynamics is the study of the motion of non-deformable, free-moving bodies through space. Such setups involving rigid bodies are prevalent throughout physics courses, leading to the value of such a simulation. The engine will improve upon many current models by focussing on distinctly accurate mathematic techniques. The two modern techniques that will be developed and investigated for optimizations throughout this project are Runge-Kutta 4 integration – iterative methods developed by German mathematicians to approximate differential equations – and the Separating Axis Theorem, a mathematical theorem used to detect collisions of convex objects lying on a 2-dimensional plane. The engine interface will allow for the speedy input of a variable number of rigid body and interaction mechanisms, specifically optimized for an educational environment.

Keywords: rigid-body dynamics, physics, Runge-Kutta 4 integration, Separating Axis Theorem, simulation

1 Introduction

The proper visualization of physical models is required to properly understand physics concepts, yet such visualizations can be difficult to produce

without the assistance of concrete examples. It is often not possible to simulate models using actual physical components in the classroom, and computers can therefore be substituted as a valuable tool towards allowing students visualize physics setups. This project will design such a physics simulation, specifically targeted towards physics students for classroom use. The use of computer simulation engines in academic physics environments has many further advantages. Precise results can be obtained from computers with almost no error margin, and the model can be examined from a number of different perspectives over varying time intervals to obtain results. The goal of this project is to develop such a physics engine, so that physical setups can be visualized on a computer with maximum accuracy, and the complex interactions between rigid bodies and other physical objects can be studied. The engine will focus on 2-dimensional space, due to its common use in introductory physics courses and its consistencies with 3-dimensional space. The Runge-Kutta 4 integration methods and the Separating Axis Theorem for collision detection will be primarily researched fully implemented to improve the simulation over many current models.

2 Background

Physics engines present the unique problem of combining small-scale rigid-body oriented interactions with large-scale physics engine scalability. Research into rigid body dynamics engines can be broken down into the microcosmic lower-level interactions between rigid bodies, and the macrocosmic large-scale design structures of physics engines. On the microcosmic level, the simulation uses the Runge-Kutta 4 methods of integration and the Separating Axis Theorem to calculate impulse forces and collisions. On the macrocosmic level, the model-view-controller design is used by the engine to separate collision detection algorithms and other interactions from the rest of the engine.

2.1 Runge-Kutta 4 Integration Methods

Runge-Kutta 4 integration is a method for solving differential equations to derive the physical state of a moving rigid body. As a simple example, variable acceleration functions are integrated using the Runge-Kutta 4 methods to derive the velocities of bodies. The traditional method for performing

such calculations, and the method used by many physics engines today, is Euler's method, where slope estimates are made using the particular state of the differential equation at the current time step. The error margin of Euler's method grows increasingly larger over larger time intervals, to the point where such simulations cannot be used to extract accurate numerical results after a short time. Runge-Kutta 4 integration, developed by the German mathematicians Carl Runge and Martin Kutta in the early twentieth century, significantly improves upon this technique by using previous iterations to calculate a much more accurate slope estimate. The previous slope function are summed over the past six time steps at their midpoints, the result is divided by the total number of distinct slope values, and the estimate (called h) is multiplied by the time interval (in milliseconds) to calculate the following integration.

2.2 Collision Duplication Prevention, Efficiency, and Modularization

One of the primary challenges of this project is to develop an optimized collision detection system that does not result in malformed or duplicate collisions. A number of mathematical and computer science techniques have been developed to implement such a system. This project uses a hash set to store collision objects for each collision detection. The hash prevents duplicate Collisions from being calculated, and it allows for insertions and deletions to be performed in O(1) time, allowing for a significant collision detection speedup. Perhaps most importantly, using Collision objects stored in a HashSet allows me to completely separate the collision detection code from the rest of my simulation. One of the original goals of my engine was to make it scale easily, and by separating the interaction code from the rest of my engine, I easily add physical interactions and mathematical techniques at will.

2.3 The Separating Axis Theorem

The Separating Axis Theorem is the most popular method of 2-Dimensional collision detection, due both to its versatility and its ease of use. The theorem states that tow convex 2-Dimensional bodies are only colliding if all of their separating axes, the axes projected onto the line that lies between the two bodies, are intersecting. Objects are able to collide in my project if they

implement the Collideable interface and the getVertices() method, which returns vectors pointing to the body's vertices. As long as these guidelines are followed, the separate axis theorem can be used to calculate the collisions between any two bodies.

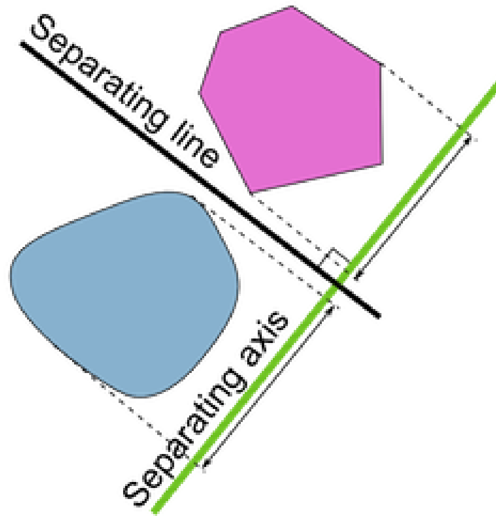

Figure 1: A diagram illustrating the what is meant by the separating line and the separating axis, the two defining parts of the Separate Axis Theorem.

2.4 *a posteriori* versus *a priori* Collisions

Another aspect of collision detection that must be worked out is whether to use *a posteriori* or *a priori* to detect collisions. The two methods are summarized below:

1. *a posteriori* Under this method of collision detection, objects are first advanced in the simulation (i.e. moved or modified), and then the physics engine checks for collisions. This is a simple method of detecting collisions, as it requires very few advance calculations to detect if the collisions took place. However, it is then more difficult to determine how the objects should react to the collisions.

2. *a priori* With this method, the trajectories of objects are accurately calculated, and the engine then detects collisions. This makes the col-

lision detection significantly harder to develop, the reactions of objects to collisions are much easier to model.

Although a priori collision detection eliminates a few problems associated with collision response, a posteriori collision detection is more representative of natural collisions, and it allows for easier scaling and separation of the collision detection algorithms. Thus, my project uses a posteriori collision detection. The simulation stores the states of objects in the previous time step to determine the magnitude of collision responses.

3 Development

I am using the Model-View-Controller (MVC) programming paradigm to develop my project. The architectural program is used by a number of software systems, such as Microsoft's .NET framework, the open source Ruby on Rails framework, and a number of common software platforms, such as the Facebook application platform and twitter. The MVC design differs from the typical two-part design of physics simulators, where the interface is separated from the simulator, by adding a third module: the model, used to represent the underlying data of the engine.

3.1 The Model-View-Controller Application Architecture

The Model, where project data (i.e. the underlying physical bodies) is stored, is represented by a series of classes in the model Java package. The Controller, which calculates the interactions between the physical components and passes their references to the GUI, is housed in the controller Java package. Lastly, the View, which is composed of the physics engine's GUI, is contained in a number of Java Swing classes in the view Java package.

The Controller (the heart of the physics engine) was first developed by coding a framework to easily add physical interactions to the engine. Directly following, development was begun on tension interactions and the Separating Axis theorem, so that these algorithms could be swiftly incorporated into the controller. After all the interactions had been coded into the controller, the model was built, so that physical objects could be added to the simulation. Lastly, the view was coded and interfaced with the controller, so that a GUI would be available to display the graphics of my project.

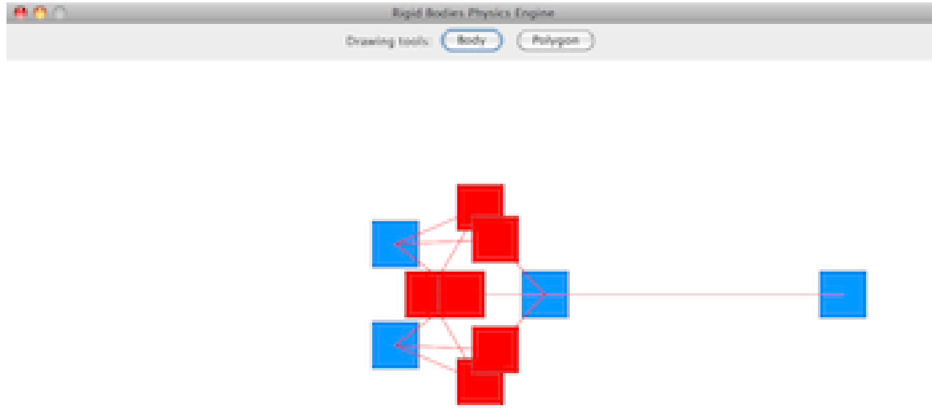

Figure 2: A screenshot of the physics engine in its current form. Ten bodies are connected via springs, and the simulation is run over a time step. Colliding bodies, as detected by the Separate Axis Theorem, are shown in red.

The interface has been developed using common application interface guidelines to create a flowing, easy-to-use, and academic user interface. Objects can be drawn using simple drawing tools, and their attributes will soon be editable using an editing pane attached to the right side of the drawing window. Such interfaces are common amongst other academic software platforms, allowing the interface to be easily understood and used by physics students.

4 Expected Results

The primary output that will be extracted from the physics engine are numerical results pertaining to the physical states of a simulation after a time step. A number of online physical models, in addition to textbook setups, are available to test the numerical results of my simulation. Furthermore, the graphics displayed by my physics engine can be evaluated for their resemblance to realistic physical phenomena.

One of the most important aspects of my simulation will be its effective-

ness in an educational setting. To test the straightforwardness and academic flexibility of my program, I will employ the use of physics students to draw their physical models into the application, after which the models can be evaluated for their similarity to the models portrayed in the textbook. The graphical user interface will be successful if is is fully functional, expandable, straightforward, and easy to use. The time it takes to draw physical models into the application will be measured. Any additional tools required to solve physics setups will be added, and the measuring capabilities required to extract proper physical information will be added. If the physics engine is fully functional and accurate, in addition to providing an educational and straightforward interface, then my project will be a success.

References

[1] C. Glocker, "On Frictionless Impact Models in Rigid-Body Systems", http://www.jstor.org/stable/3066369

[2] D. E. Stewart, "Rigid-Body Dynamics with Friction and impact", http://www.jstor.org/stable/2653374

[3] R. K. Alexander, J. J. Coyle, "Runge-Kutta Methods and Differential-Algebraic Systems", http://www.jstor.org/stable/2157857

[4] O. Hilliges, S. Izadi, D. Kirk, A. Garcia-Mendoza, A. D. Wilson, "Bringing Physics to the Surface", http://portal.acm.org/citation.cfm?id=1449715.1449728

[5] C. B. Price, "The usability of a commercial game physics engine to develop physics educational materials: An investigation", http://portal.acm.org/citation.cfm?id=1401790.1401794